Proceedings of the Third International Workshop on Sustainable Ultrascale Computing Systems (NESUS 2016) Sofia, Bulgaria

Jesus Carretero, Javier Garcia Blas, Svetozar Margenov
(Editors)

October, 6-7, 2016

Alonso, P., Reddy Manumachu, R. & Lastovetsky, A. (2016). Heterogeneous computation of matrix products. En *Proceedings of the Third International Workshop on Sustainable Ultrascale Computing Systems (NESUS 2016) Sofia, Bulgaria* (pp. 51-58). Madrid: Universidad Carlos III de Madrid. Computer Architecture, Communications, and Systems Group (ARCOS).

Heterogeneous computation of matrix products

Pedro Alonso
Universitat Politècnica de València, Spain
palonso@upv.es

Ravi Reddy Manumachu
University College of Dublin, Ireland
ravi.manumachu@ucd.ie

Alexey Lastovetsky
University College of Dublin, Ireland
alexey.lastovetsky@ucd.ie

Abstract

The work presented here is an experimental study of performance in execution time and energy consumption of matrix multiplications on a heterogeneous server. The server features three different devices: a multicore CPU, an NVIDIA Tesla GPU, and an Intel Xeon Phi coprocessor. Matrix multiplication is one of the most used linear algebra kernels and, consequently, applications that make an intensive use of this operation can greatly benefit from efficient implementations. This is the case of the evaluation of matrix polynomials, a core operation used to calculate many matrix functions, which involve a very large number of products of square matrices. Although there exist many proposals for efficient implementations of matrix multiplications in heterogeneous environments, it is still difficult to find packages providing a matrix multiplication routine that is so easy to use, efficient, and versatile as its homogeneous counterparts. Our approach here is based on a simple implementation using OpenMP sections. We have also devised a functional model for the execution time that has been successfully applied to the evaluation of matrix polynomials of large degree so that it allows to balance the workload and minimizes the runtime cost.

Keywords Matrix multiplication, heterogeneous system, energy consumption, matrix polynomials

I. Introduction

Matrix multiplication is one of the most essential computational kernels used in the core of scientific applications. This operation has been highly studied in the past in order to improve the efficiency of its computation in both sequential and parallel computer architectures. It has also received full attention in parallel heterogeneous environments. Many contributions in this context basically propose irregular partitions of the factor matrices that can efficiently be mapped on the computing resources; see for instance [10, 16, 12].

It is difficult to find actual implementations of the matrix multiplication on heterogeneous nodes that feature very different devices. The MAGMA project, for instance, aims to develop a dense linear algebra library similar to LAPACK but for heterogeneous/hybrid architectures; it is one of the most active projects that implement BLAS routines for nodes featuring accelerators [22]. Currently, MAGMA implements a version for NVIDIA GPUs in which the matrix multiplication is carried out only by the GPUs, i.e. the CPU does not intervene. The MAGMA project also provides with a version, MAGMA MIC, which provides hybrid algorithms that involve the host CPU and one or more Intel Xeon Phi processors. However, this project does not use both NVIDIA GPUs and MICs processor all together in the same host. Authors of [13] propose a programming model for heterogeneous computers featuring CPU, a GPU and a Xeon Phi with the aim to incorporate it to MAGMA library. However, they have not shown its proposal with matrix multiplication. Hence and as a starting point, we propose here a simple implementation to carry out parallel heterogeneous matrix multiplications in a node composed by CPU cores, one NVIDIA GPU, and one Intel Xeon Phi.

As it is explained in the next section, in this paper we are interested in evaluating matrix polynomials of only square matrices. Section III shows the application implemented to carry out a square matrix multiplication on these three different devices. The following section shows experimental results both in time and energy consumption of our application. In Section V we propose a model to implement a heterogeneous matrix multiplication routine that can exploit easily the underlying hardware. We finish the paper with some conclusions and proposals for future research.

II. Matrix polynomials

An application for matrix multiplications is, for instance, the calculus of matrix polynomials. Matrix polynomials are used, e.g. for the computation of functions of matrices [9] by the Taylor method. A matrix function is the exponential of a matrix [21]. This function appears in the solution of many engineering and physics phenomena which are governed by systems of linear first-order ordinary differential equations with constant coefficients [15]. Also, the matrix exponential appears in other scientific contexts like, e.g. control theory [14] or theory of multimode electric power lines [24]. Some other engineering processes are described by second order differential equations, whose exact solution is given in terms of the trigonometric matrix function sine and cosine [11, 17].

There are different techniques for computing or approximating matrix functions. Some of them are very general but others are specialized to particular functions. Two techniques are widely used to approximate a matrix function, one is based on polynomial approximations and the other is based on rational approximations. The one based on polynomial approximations makes intensive use of matrix multiplications. For example, the matrix exponential can be calculated efficiently by using Taylor series [21], which is in turn formulated as a matrix polynomial. Other trigonometric matrix function is the cosine of a matrix. This function has been tackled in [20] to show that it is possible to perform its computation in a

Algorithm 1 Algorithm for the evaluation of a matrix polynomial.

```
1: function EVALUATE( n, X, d, α̃ ) return P
2:     P ← α_0 I
3:     P ← P + α_1 X
4:     B ← X
5:     for i ← 2, d do
6:         A ← B
7:         B ← X · A
8:         P ← P + α_i B
9:     end for
10: end function
```

very efficient way by using also a Taylor series approximation. As it has been shown in [19] it is possible to obtain more accuracy with polynomial approximations than with rational approximations with similar or even lower computational cost. Another advantage of using this technique is due to the fact that the most expensive operations are all matrix products and there exist many libraries that provide efficient implementations of this operation in different environments. For instance, one can find very optimized implementations for multicore processors in Intel MKL, OpenBLAS, or BLIS. Another example is CUBLAS, a library that includes a very efficient routine to perform matrix multiplications in NVIDIA GPUs. This library was recently used in [9] to implement the algorithm proposed in [20] that computes the cosine of a matrix using one or two GPUs.

A matrix polynomial P of degree d can be defined as

$$\begin{aligned} P &= \sum_{i=0}^{d} \alpha_{d-i} X^{d-i} \qquad (1) \\ &= \alpha_d X^d + \alpha_{d-1} X^{d-1} + \cdots + \alpha_1 X + \alpha_0 I \,, \end{aligned}$$

where $X, I \in \mathbb{R}^{n \times n}$, being I the identity matrix. The polynomial matrix X can be arbitrary large, e.g. when appears in the solution of PDEs related to fluid dynamics. Also the polynomial degree d can be very large, e.g. 30 is a common number in the calculus of a trigonometric matrix function [19].

In theory, the evaluation of a matrix polynomial with the form (1) is quite straightforward by using, for instance, Algorithm 1. There exist algorithms that allow to reduce the total number of matrix products by means of the so called Paterson–Stockmeyer method [9]. However, these methods also need an efficient implementation of the matrix product. In any case, the efficiency of the evaluation of a matrix polynomial depends on how efficient the underlying matrix multiplication routine is. When the computational resources are the cores of a multicore, we rely on *threaded* routines (e.g. Intel MKL) that exploit all the CPU cores concurrently to perform this computation transparently to the user. But things are more complicated in a heterogeneous environment, where the computational resources are different among them and, in turn, are "far away" from the main memory where data initially reside.

III. A Hybrid Matrix Multiplication Application

In order to solve efficiently problems like the evaluation of matrix polynomials that intensively use matrix multiplications on a heterogeneous server, we propose an implementation for a matrix multiplication application and present an experimental study of its performance in both execution time and energy consumption.

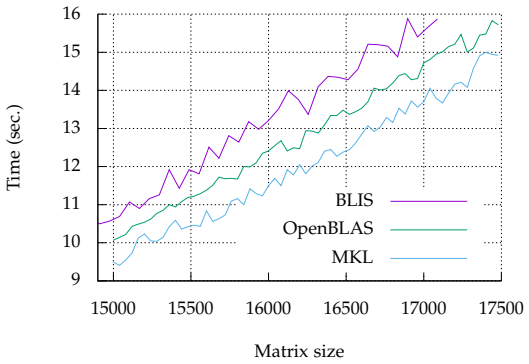

Figure 1: Execution time for a square matrix multiplication in one core using different libraries.

III.1 The Hardware and Software Used

The heterogeneous server we have been working with features the following devices:

- **CPU**: Two sockets with an Intel Xeon CPU E5-2670 v3 at 2.30 GHz each. This processor has 12 cores so the server contains a total of 24 threads. The main memory of the host is 64 GB.
- **GPU**: NVIDIA Tesla K40c with 2880 cores and 12 GB of device memory.
- **PHI**: An Intel Xeon Phi 3120A coprocessor with 57 processors (228 cores) and 6 GB of device memory.

Although the term "device" is usually assigned to accelerators only, i.e. GPU and PHI, in the next and for the sake of simplicity, we will use it to denote the three of them.

On the software side, we have within reach different implementations of BLAS [1] to perform the matrix multiplication:

MKL: Intel Math Kernel Library is an optimized implementation of linear algebra routines contained in BLAS and LAPACK, and other mathematical functions like the FFT. This library is available for multicore x86 processors, and also for the Intel Xeon Phi coprocessor [3]. There exist "threaded" routines, e.g. the matrix multiplication routine `GEMM`, for both devices.

OpenBLAS: OpenBLAS is an optimized BLAS library based on GotoBLAS2 1.13 BSD version [4]. Used in the CPU.

BLIS: This library is self-described as "a portable software framework for instantiating high-performance BLAS-like dense linear algebra libraries". In addition the "framework was designed to isolate essential kernels of computation that, when optimized, immediately enable optimized implementations of most of its

commonly used and computationally intensive operations" [23]. The library is accessible in [2]. It has been used in the CPU.

CUBLAS: BLAS implementation for NVIDIA GPUs [8].

We performed a simple experimental analysis of the speed of the matrix multiplication (`GEMM`) in the CPU (Figure 1). For this test we used the maximum available CPU cores, i.e. 24. (We ignored the fact that Hyper-Threading (HT) can be enabled to give a total of 48 logical processors. We observed that using just one thread per core is enough to fully exploit the execution resources of the core and not increase in performance can be achieve by activating HT.) It must be said that the performance of BLIS could be probably better by selecting the best parallel configuration. Contrary to the other two packages, BLIS is tuned by setting the value of up to four environment variables. That value corresponds to the number of threads that will be used to parallelize a given loop among the five nested loops in which the matrix multiplication is implemented in order to exploit the hierarchical set of intermediate memories of the most current architectures. In this test, only the outer loop was parallelized. A more suitable combination of values are likely to produce a better performance of BLIS, however, we decided not to test the large set of different combinations with the idea that barely the performance would outperform MKL in this machine. Consequently, we consider the performance of Intel MKL to be the best and, therefore, it is the only library used on the CPU side.

III.2 Implementation option

To proceed towards a heterogeneous matrix product, we started by implementing an application that partitions the problem into three concurrent pieces so that the three devices can cooperate in the solution. There exist different options to implement such an application. However, all the options can be gathered into two main classes standing for the use of light processes (threads), or heavy processes. The last option can be implemented e.g. by using MPI [7]. Here, we decided to use a simple approach based on threads, which are spawned by means of OpenMP sections.

The application has been implemented with OpenMP sections, so that each device code is included in a given section (Listing 1). The code for the Intel Xeon Phi, in lines 31–32, is implemented in a different source file (Listing 2) and compiled separately. This is because it is necessary to compile this code with the Intel C compiler (`icc`). For the compilation of the rest of the C code of the application we used the GNU compiler (`gcc`) since there exists incompatibility between the available versions for the NVIDIA compiler (`nvcc`, version 7.5) and for the Intel compiler (`icc`, version 16.0).

The basics of the heterogeneous multiplication are easy. To perform the multiplication $C = AB$, matrix A is completely broadcast to the two accelerators from the Host computer. Matrix B, however, is partitioned into three blocks of consecutive columns. The second block is uploaded to the GPU, the third one is uploaded to the PHI, and the first one remains into the host memory. The amount of columns of each block is denoted in Listing 1 by the values of variables `gpu_n`, `phi_n`, and `cpu_n` for the GPU, the PHI, and the CPU, respectively. Currently, the application receives these values as arguments by command line, in particular, the user sets the percentages for the GPU and for the PHI in the range $[0, 1]$, the rest is computed by the CPU. Upon termination of the execution,

```
1  int gpu_n  = (int) (gpu_weight * n);
2  int phi_n  = (int) (phi_weight * n);
3  int cpu_n  = n-gpu_n-phi_n;
4  #pragma omp parallel sections num_threads(3)
5  {
6  #pragma omp section
7  { // GPU
8    if( gpu_n ) {
9    cublasHandle_t handle;
10   CUBLAS_SAFE_CALL( cublasCreate(&handle) );
11   double *gpu_A, *gpu_B, *gpu_C;
12   CUDA_SAFE_CALL( cudaMalloc((void **) &gpu_A, n*n*sizeof(double) ) );
13   CUDA_SAFE_CALL( cudaMalloc((void **) &gpu_B, n*gpu_n*sizeof(double) ) );
14   CUDA_SAFE_CALL( cudaMalloc((void **) &gpu_C, n*gpu_n*sizeof(double) ) );
15   CUBLAS_SAFE_CALL(cublasSetMatrix(n, n, sizeof(double), A, n, gpu_A, n ));
16   CUBLAS_SAFE_CALL( cublasSetMatrix( n, gpu_n, sizeof(double),
17                                      &B[n*cpu_n], n, gpu_B, n ) );
18   CUBLAS_SAFE_CALL( cublasDgemm(handle, CUBLAS_OP_N, CUBLAS_OP_N, n, gpu_n,
19                         n, &alpha, gpu_A, n, gpu_B, n, &beta, gpu_C, n ) );
20   CUBLAS_SAFE_CALL( cublasGetMatrix( n, gpu_n, sizeof(double), gpu_C, n,
21                                      &C[n*cpu_n], n ) );
22   CUDA_SAFE_CALL( cudaFree(gpu_A) );
23   CUDA_SAFE_CALL( cudaFree(gpu_B) );
24   CUDA_SAFE_CALL( cudaFree(gpu_C) );
25   CUBLAS_SAFE_CALL( cublasDestroy(handle) );
26   }
27 }
28 #pragma omp section
29 { // PHI
30   if( phi_n ) {
31   gemmPHI( n, phi_n, n, alpha, A, n, beta, &B[n*(cpu_n+gpu_n)], n,
32            &C[n*(cpu_n+gpu_n)], n );
33   }
34 }
35 #pragma omp section
36 { // CPU
37   if( cpu_n ) {
38   dgemm( &transa, &transb, &n, &cpu_n, &n, &alpha, A, &n, B, &n,
39                                                    &beta, C, &n );
40   }
41 }
42 }
```

Listing 1: Code for the heterogeneous matrix multiplication.

the resulting matrix C appears partitioned and distributed among the three devices. We include in the application, and in the time measurement, the operation of gathering the result in the memory location allocated into the host to store the resulting matrix.

The code for the execution in the GPU is quite regular (Lines 7–27). It includes creation of the CUBLAS context, allocating memory for the three matrix factors, uploading matrices, executing the matrix product, downloading the result, and freeing the resources involved in the computation.

For the Xeon Phi, we used the "offload mode" of computation, that is, data is explicitly uploaded to the device and the operation is also explicitly executed there. Thus, the programmer have control of what exactly is executing the coprocessor. Arguments `in`, `out`, and `inout` specify clearly the direction of variables characterized by those words. The operation is actually performed by calling to the BLAS matrix multiplication routine using the MKL version.

Finally, the code executed by the CPU only includes a call to the `gemm` routine (lines 38–39) for the matrix computation using MKL as well. We used the `fortran` interface instead of the C one used for the PHI for no specific reason but the application is oblivious of this.

Attention must be paid to the way in which the application is executed in our heterogeneous server. As it has been implemented, only three OpenMP threads are created so that each one will execute a different section. There will be, thus, one thread bound to each accelerator for data transference and control purposes. For the CPU case, however, the execution of the MKL routine will use only

```
void gemmPHI( int m, int n, int o, double alpha, double *A, int lda,
              double beta, double *B, int ldb, double *C, int ldc ) {
#pragma offload target(mic) in(m,n,o,alpha,beta,lda,ldb,ldc) \
                 in(A:length(m*o)) in(B:length(o*n)) inout(C:length(m*n))
{
  cblas_dgemm(CblasColMajor, CblasNoTrans, CblasNoTrans, m, n, o,
                             alpha, A, lda, B, ldb, beta, C, ldc );
}
}
```

Listing 2: Code for the heterogeneous matrix multiplication in the Xeon Phi (offload mode).

one thread. To use more threads (cores) collaborating in the matrix multiplication on the CPU side, the "nested parallelism" ability must be explicitly set. In addition, there are more environment variables that control the behaviour of the application (Table 1).

This is an example of execution:

```
shell_$ MKL_NUM_THREADS=22 OMP_NESTED=TRUE MKL_DYNAMIC=FALSE
        MKL_MIC_ENABLE=0 MIC_ENV_PREFIX=MIC
        MIC_KMP_AFFINITY=balanced,granularity=fine
        MIC_OMP_NUM_THREADS=228 MIC_USE_2MB_BUFFERS=64K
        numactl --physcpubind=+0-11,12-23 program 10000 0.48 .15
```

The example executes the program `program` which generates two random matrices of order $n = 10000$. The GPU performs 48% of the computation, 15% is carried out by the PHI, and the rest, 37%, is computed by the CPU.

It should also be noted that the server has the hyperthreading enabled, but we decided not to use all the 48 threads and always use 24 as a maximum number of threads instead. For instance, when operating with the three devices, two threads are bound to one accelerator each, leaving the other 22 for the execution of the matrix multiplication in the CPU.

In addition, we have always used core affinity. This is to prevent threads from leaping amongst the cores at runtime, so as to reduce the variability of the execution times and also to improve the performance of all the devices attached to the host. Concretely speaking, we use the tool `numactl` to bind threads to cores.

The following is an example of the output of the application:

```
n = 10000 (CPU = 38.00% GPU = 50.00% PHI = 12.00%) [4 reps]
          (cpu = 1.17 sec. gpu = 1.14 sec. phi = 1.19 sec.)
          (1.30 sec. 1541.47 gflops)
```

for a random matrix of size $n = 10000$. The weight used for each device in this example results in a workload rather well balanced.

III.3 Energy consumption

We are also interested on evaluating the energy consumption of the devices participating in the matrix multiplication with the aim at, first, understanding the power trace of each device and, second, exploring a workload distribution which can result in energy savings.

For the energy measurement, we have used a tool called `powerrun` [5]. This tool is a wrapper to other tools for measuring the power draw of the CPU (uses PAPI and Intel PCM), of the GPU (uses NVML [18]), and of the PHI (uses Intel's MPSS [6]). The tool gathers the power samples of all the devices under operation and dumps a power trace to a file to compute the energy consumed during the execution time. This tool provides a library to instrument the code under test with simple calls that frame the part of the code to be measured.

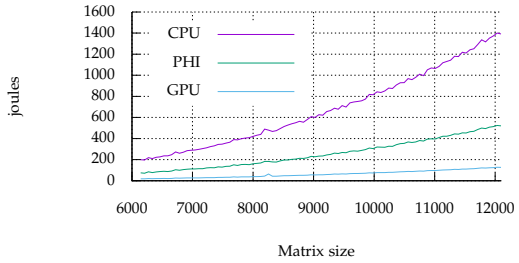

Figure 2: Energy consumption when executing a matrix multiplication in the CPU and the other two devices remain idle.

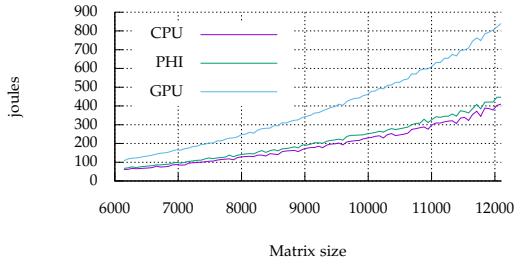

Figure 3: Energy consumption when executing a matrix multiplication in the GPU and the other two devices remain idle.

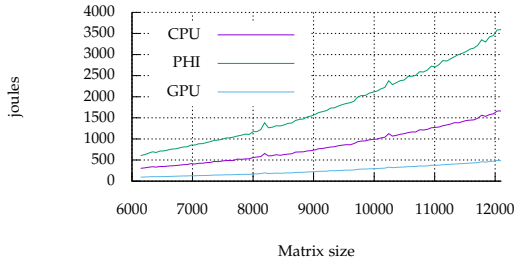

Figure 4: Energy consumption when executing a matrix multiplication in the PHI and the other two devices remain idle.

We provide here three tests that show the energy consumption (in joules) of the three devices, respectively. In each test, the three devices are operating concurrently. Only one of them is working on a matrix multiplication while the other two remain idle. The test samples the energy of the three devices.

Figure 2 shows the energy consumed by the system when only the CPU is "working" and is rather easy to interpret. The CPU is the most consuming device since it is the only one that performs useful work, while the other two consume the energy in idle state. It is also quite clear the difference in energy consumption when idle between the two accelerators, being very low in the case of the NVIDIA GPU

Variable name	Meaning
`OMP_NESTED`:	Set to `TRUE` to ensure that MKL uses more than one thread when called inside an OpenMP section.
`MKL_NUM_THREADS`:	Number of threads used by MKL (CPU).
`MKL_DYNAMIC`:	Set to `FALSE` to avoid MKL automatically selects the number of threads (CPU).
`MKL_MIC_ENABLE`:	Set to 0 to avoid the Xeon Phi is used to accelerate the CPU computation.
`MIC_ENV_PREFIX`:	Specifies the environment variables with prefix `MIC` will address only the PHI.
`MIC_OMP_NUM_THREADS`:	Number of threads used by the PHI to execute MKL routines.
`MIC_KMP_AFFINITY`, `MIC_USE_2MB_BUFFERS`:	These variables control the efficiency of the Xeon Phi in the execution of the matrix multiplication routine. They have been set to such values according to the advice of Intel documentation.

Table 1: Meaning of shell variables used to execute the heterogeneous matrix multiplication application.

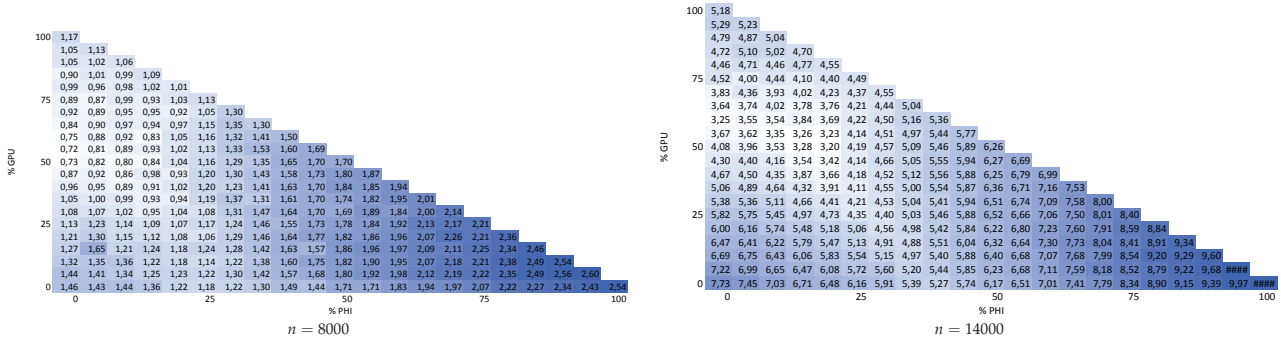

Figure 5: Execution time in seconds for a matrix product varying the weight of workload on each device.

compared with the Intel Xeon Phi.

Figure 3 shows the energy consumption when the GPU is the only device operating on a matrix multiplication. Note that one of the cores of the CPU is also working since it is in charge of sending the two matrices to be multiplied and receiving the resulting one. The consumption of the Intel Xeon Phi is very large in idle state when compared with the CPU.

As expected, the consumption of the Intel Xeon Phi is quite large when executing the matrix multiplication (Figure 4). Also in this case one of the cores of the CPU is working to feed the coprocessor with the two factor matrices and to receive the solution matrix.

IV. Experimental Results of the Matrix Multiplication Application

Figure 5 shows the execution time in seconds spent by the application to perform a matrix multiplication of two square matrices of sizes $n = 8000$ and $n = 14000$. The two graphics show times for different weight combinations. The percentage of computation carried out by the GPU is shown on the y-axis, while the work done by the PHI is shown on the x-axis. These two values are selected by the user. The rest of the computation is performed by the CPU. The figure shows less execution times (clearer cells) within the region between $\approx 25\%$ and $\approx 50\%$ for the GPU, and $\lessapprox 20\%$ for the PHI in the case of the problem sizes selected. There exists more opportunity for the PHI to participate as long as the problem size increases.

Figure 6 shows the percentages of the minimum values obtained for the problem sizes $n = 8000, 10000, 12000, 14000$, which are 0.72 sec., 1.30 sec., 2.08 sec., and 3.20 sec., respectively. For large problems

	$n \in [2000, 10000]$	$n \in [10000, 14000]$
$w_{\text{phi}} =$	0	$\frac{n}{200} - 50$
$w_{\text{gpu}} =$	$\frac{n}{800} + 47.5$	$-\frac{n}{400} + 85$
$w_{\text{cpu}} =$	$100 - w_{\text{gpu}}$	$100 - (w_{\text{phi}} + w_{\text{gpu}})$

Table 2: Functions of the weight for each device for the execution time of the matrix multiplication.

both the CPU and the GPU reduce their weight to make room for the PHI, which does not contribute to the task with any size smaller than $n = 12000$. We can approximate the weight of each device, i.e. w_{cpu}, w_{gpu}, and w_{phi}[1], by the two linear functions shown in Table 2 for two intervals. By means of a larger experimental setup we could easily devise a functional model that allows to predict the best percentage of workload to be mapped on each device. However, we must take into account that there exist a problem size not very much smaller than $n = 2000$ for which it is not worthwhile to use the GPU. Also, for problem sizes $n > 14000$, the weight to be assigned to each device stabilizes around a fix value ($w_{\text{phi}} \approx 15\%$ and $w_{\text{ghi}} \approx 55\%$). However, as the problem size increases a little more, out-of-core algorithms are required and these functional models can significantly change.

Things are slightly different when we observe the total energy consumed by the matrix multiplication application. The minimum values of energy (in joules) are 379, 700, 1177, and 1783, for the problem sizes 8000, 10000, 12000, and 14000, respectively. Figure 7

[1]Note that the number of matrix columns assigned to a device d is $n_{\text{d}} = n \cdot w_{\text{d}}$, where $\text{d} = \text{cpu}, \text{gpu}, \text{phi}$.

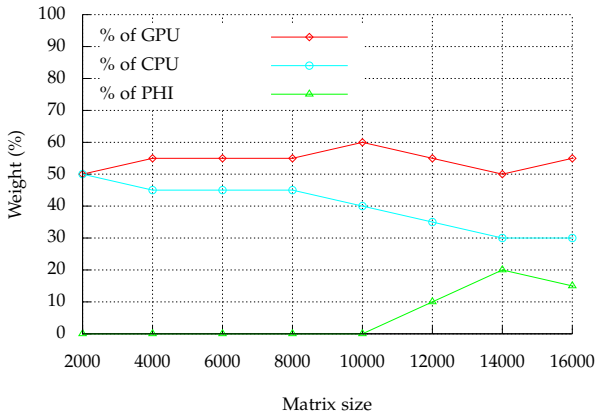

Figure 6: Functional model in graphics for the execution time of the matrix multiplication.

shows, as an example, the energy consumption with problem sizes $n = 8000$ and $n = 14000$. The corresponding weights of w_{gpu} in which we can find these minimum values are 55%, 60%, 60%, and 60%, for each problem size, respectively, and 0% for w_{phi}. These numbers show that while Intel Xeon Phi can contribute a little to reduce the execution time, it can contribute nothing towards reducing the energy consumption in any case, as it was expected according to Figures 2–4. The NVIDIA GPU is, currently, a more efficient device for HPC. In this particular server and for large problem sizes ($n > 10000$), the Intel Xeon Phi is used as a trade-off between execution time and total energy consumption.

We also figured out the dynamic energy of the application, i.e. the energy due to the execution of the application and that we obtain after taking away the energy consumed by each device in idle state. The results showed that we can find the minimum value for the dynamic energy for all problem sizes when none of the accelerators are used. This is due, on one hand, to the high energy consumption of the PHI and, on the other hand, to that the NVIDIA GPUs has two different performance states (when idle) that are difficult to control and disturb the actual energy measurement when the device is idle.

V. A Heterogeneous Matrix Multiplication System for Evaluating Matrix Polynomials

For this part of the work we have got a representative application of matrix polynomials, i.e. that intensively uses matrix multiplications. This application has been recently developed and it allows to compute the cosine of a matrix [9]. Implementing this application on the top of a heterogeneous matrix multiplication routine allows to get the most out of a heterogeneous computer.

The task of developing a program that solves this problem poses a big challenge from the performance point of view, as we showed before, but also from the programmability point of view. Thus, in order to make as easy as possible the programming task we propose a system based on three different kind of objects which

Algorithm 2 Heterogeneous algorithm for the evaluation of a matrix polynomial.

1: **function** EVALUATE(n, X, d, $\bar{\alpha}$) **return** P
2: $P \leftarrow \alpha_0 I$
3: $P \leftarrow P + \alpha_1 X$
4: $P_D \leftarrow P$
5: $X_R \leftarrow X$
6: $B_D \leftarrow X_D$
7: **for** $i \leftarrow 2, d$ **do**
8: $A_D \leftarrow B_D$
9: $B_D \leftarrow X_R \cdot A_D$
10: $P_D \leftarrow P_D + \alpha_i B_D$
11: **end for**
12: **end function**

represent matrices. Let M be a square matrix stored into the host main memory, then we report the following definitions:

- *Regular matrices*: these matrices are uniquely stored into the CPU main memory, e.g. the matrix M itself.
- *Replicated matrices*: these matrices, denoted by subscript R (e.g. M_R), are replicated into all the three devices.
- *Distributed matrices*: these matrices, denoted by subscript D (e.g. M_D), are partitioned in column blocks and scattered into all the three devices.

We use these objects to rewrite Algorithm 1 into its heterogeneous counterpart, Algorithm 2 . In the heterogeneous algorithm, each matrix object is characterized by its condition according to where the entries of this matrix are stored, i.e. *regular*, *replicated* or *distributed*.

We also describe the communication operation that takes place between each pair of matrix-types as follows:

- $M \rightarrow M_R$: This is a *Broadcast* communication.
- $M \rightarrow M_D$: This is *Scatter* communication.
- $M_R \rightarrow M$: This is a dummy operation since the destination matrix is already into CPU memory and can be implemented through a local copy.
- $M_R \rightarrow M_D$: This is a local copy of the right data partition.
- $M_D \rightarrow M$: This is a *Gather* communication.
- $M_D \rightarrow M_R$: This is an *Allgather* communication.

We make the assumption that there exists just one distribution for all the distributed matrices involved, and this distribution, represented by the tuple $(w_{\text{cpu}}, w_{\text{gpu}}, w_{\text{phi}})$, has been previously calculated and it is known before executing the algorithm. There are conversion operations between matrix types in steps 4 and 5. Steps 6 and 8 are local copies of the proper data objects, and Step 10 can be readily implemented calling to the BLAS `saxpy` routine. Step 9 is the matrix multiplication of the replicated matrix X_R by the distributed matrix A_D, and this is just the multiplication tackled in Section III.

Figure 8 represents the evolution of runtime with regard to the polynomial degree for the evaluation of two polynomials of size

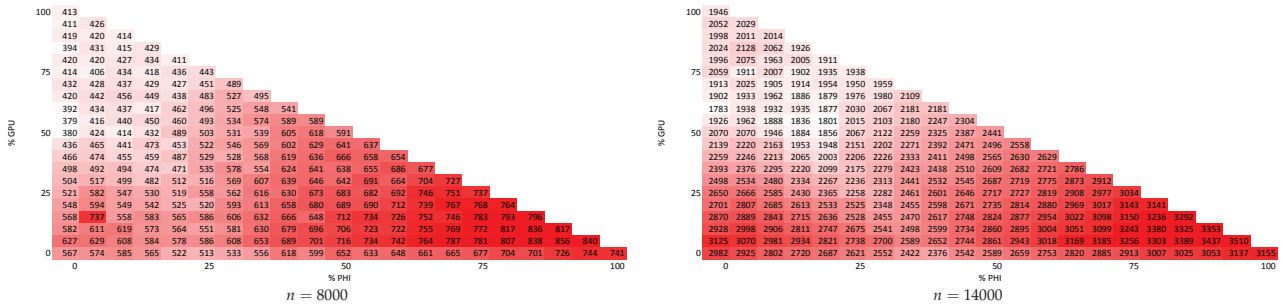

Figure 7: Energy consumption in joules for a matrix product varying the weight of workload on each device.

$n = 10000$ and $n = 14000$, respectively, of random coefficients. The figure shows the execution times using only the CPU versus using the three devices. For the second case, we selected the distribution tuple suggested by Figure 6 and Table 2 for each problem size. The figure demonstrates that the evaluation of matrix polynomials can be speeded up significantly by using all the devices in the heterogeneous platform.

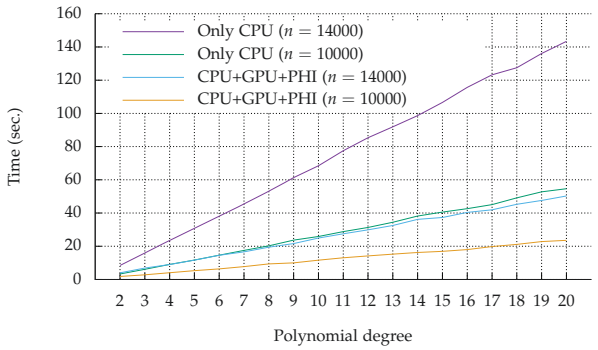

Figure 8: Execution time for the evaluation of matrix polynomials.

VI. Conclusions and future work

This work has presented an application for matrix multiplications in a heterogeneous node composed by a multicore CPU and two very different accelerators. We have shown that it is not difficult to implement the application using OpenMP sections. However, the incompatibility among compiler versions can make this task a bit cumbersome, and in addition, selecting the exact suitable value for the large number of environment variables is an arduous task that highly affects the performance of the application.

We have reduced the study of our application to a particular case in which all matrices involved are square. This case is motivated by our aim to evaluate efficiently matrix polynomials, which is the core operation to obtain matrix functions using the Taylor method. However, the study can be extended to rectangular matrices with little effort. We have developed a functional model for the runtime so that we can select the proper amount of work to do by each device. In our node, the K40 is the most speedy device, far more than the Xeon Phi, which only has opportunity to contribute to the computation on matrices larger than $n = 10000$. Furthermore, the Xeon Phi is currently the most expensive device in terms of energy consumption, and the K40 is the most energy efficient. Our study on the energy consumption resulted in a quite simple behaviour, i.e. the lowest total energy consumption is achieved when the GPU is used in a similar proportion as that selected to achieve the lowest execution time, provided the Xeon Phi is not used at all. It was impossible to obtain an accurate measure of dynamic energy due to specific behaviour of the GPU, which changes between two different performance states (when idle) in an unpredictable way.

Finally, we proposed a heterogeneous matrix multiplication system to make easy the programmability of algorithms based on a heterogeneous matrix multiplication. The system was successfully applied to obtain rapidly an application for the evaluation of matrix polynomials. We plan for the future to generalize this system so that we can perform products of the form

$$C_X \leftarrow \beta C_X + \alpha A_Y B_Z,$$

being $X, Y, Z \in \{\text{none}, \text{R}, \text{D}\}$, i.e. products that involve any type of matrix distribution.

Finally, our aim is to extend everything carried out in this work as fast and easy as possible to host the new upcoming FPGA device.

Acknowledgment

This research has been supported by EU under the COST programme Action IC1305, 'Network for Sustainable Ultrascale Computing (NESUS)'.

References

[1] BLAS: Basic linear algebra subprograms library. `http://www.netlib.org/blas`. Accessed: 2016-04-12.

[2] Blis: BLAS-like library instantiation software framework. `https://github.com/flame/blis`. Accessed: 2016-04-12.

[3] Intel Math Kernel Library (Intel MKL). `https://software.intel.com/en-us/intel-mkl`. Accessed: 2016-04-12.

[4] OpenBLAS: An optimized BLAS library. `http://www.openblas.net`. Accessed: 2016-04-12.

[5] `powerrun`: A tool to measure energy. Accessed: 2016-04-12.

[6] Intel Manycore Platform Software Stack (Intel MPSS). `https://software.intel.com/en-us/articles/intel-manycore-platform-software-stack-mpss`, 2016.

[7] Message Passing Iterface. `https://www.mpi-forum.org`, 2016.

[8] NVIDIA CUDA Basic Linear Algebra Subroutines (cublas) library. `https://developer.nvidia.com/cublas`, 2016.

[9] Pedro Alonso, Javier Ibáñez, Jorge Sastre, Jesús Peinado, and Emilio Defez. Efficient and accurate algorithms for computing matrix trigonometric functions. *Journal of Computational and Applied Mathematics*, 309:325–332, January 2017.

[10] Olivier Beaumont, Vincent Boudet, Arnaud Legrand, Fabrice Rastello, and Yves Robert. Heterogeneous matrix-matrix multiplication, or partitioning a square into rectangles: NP-completeness and approximation algorithms. In *EuroMicro Workshop on Parallel and Distributed Computing (EuroMicro'2001)*, pages 298–305. IEEE Computer Society Press, 2001.

[11] S. M. Cox and P. C. Matthews. Exponential time differencing for stiff systems. *J. of Comput. Physics, Elsevier*, 176:430–455, 2002.

[12] Ashley DeFlumere, Alexey Lastovetsky, Brett Becker, et al. Partitioning for parallel matrix-matrix multiplication with heterogeneous processors: The optimal solution. In *Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), IEEE 26th International*, pages 125–139. IEEE, 2012.

[13] Azzam Haidar, Jack Dongarra, Khairul Kabir, Mark Gates, Piotr Luszczek, Stanimire Tomov, and Yulu Jia. Hpc programming on intel many-integrated-core hardware with magma port to xeon phi. *Scientific Programming*, 23, 01-2015 2015.

[14] Nicholas J. Higham. *Functions of Matrices: Theory and Computation*. SIAM, Philadelphia, PA, USA, 2008.

[15] Marlis Hochbruck, Christian Lubich, and Hubert Selhofer. Exponential integrators for large systems of differential equations. *The SIAM Journal on Scientific Computing*, 19(5):1552–1574, September 1998.

[16] A. Kalinov and A. Lastovetsky. Heterogeneous distribution of computations solving linear algebra problems on networks of heterogeneous computers. *Journal of Parallel and Distributed Computing*, 61:520–535, 2001.

[17] Aly Khan Kassam and Lloyd N. Trefethen. Fourth-order time-stepping for stiff PDEs. *The SIAM J. on Scientific Comp.*, 26(4):1214–1233, 2005.

[18] NVIDIA. NVML Reference Manual. `https://developer.nvidia.com/nvidia-management-library-nvml`, 2013.

[19] J. Sastre, Javier J. Ibáñez, E. Defez, and Pedro A. Ruiz. Accurate matrix exponential computation to solve coupled differential. *Mathematical and Computer Modelling*, 54:1835–1840, 2011.

[20] J. Sastre, Javier J. Ibáñez, E. Defez, and Pedro A. Ruiz. Computing matrix functions arising in engineering models with orthogonal matrix polynomials. *Mathematical and Computer Modelling*, 57:1738–1743, 2013.

[21] J. Sastre, Javier J. Ibáñez, E. Defez, and Pedro A. Ruiz. Efficient scaling-squaring Taylor method for computing matrix exponential. *SIAM J. on Scientific Comput.*, 37(1):A439–455, 2015.

[22] Stanimire Tomov, Jack Dongarra, and Marc Baboulin. Towards dense linear algebra for hybrid GPU accelerated manycore systems. *Parallel Computing*, 36(5-6):232–240, June 2010.

[23] Field G. Van Zee and Robert A. van de Geijn. BLIS: A framework for rapidly instantiating BLAS functionality. *ACM Transactions on Mathematical Software*, 41(3):14:1–14:33, 2015.

[24] D. F. Williams, L. A. Hayden, and R. B. Marks. A complete multimode equivalent-circuit theory for electrical design. *SIAM J. on Scientific Comput.*, 102(4):405–423, 1997.